**Java**编码规范

**<V1.0>**

修订历史记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 版本 | 说明 | 作者 |
| 2011-12-19 | V1.0 | 创建文件 | 常育新 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

目录

[第1章 绪论 1](#_Toc312056000)

[1.1 目的 1](#_Toc312056001)

[1.2 范围 1](#_Toc312056002)

[1.3 参考资料 1](#_Toc312056003)

[1.4 概述 1](#_Toc312056004)

[第2章 代码组织与风格 2](#_Toc312056005)

[2.1 基本原则 2](#_Toc312056006)

[2.2 括号 2](#_Toc312056007)

[2.3 缩进 3](#_Toc312056008)

[2.4 空白行 3](#_Toc312056009)

[2.5 空格 3](#_Toc312056010)

[2.6 长度 4](#_Toc312056011)

[2.7 行宽 4](#_Toc312056012)

[2.8 换行 4](#_Toc312056013)

[2.9 对齐 7](#_Toc312056014)

[第3章 命名规范 7](#_Toc312056015)

[3.1 基本原则 7](#_Toc312056016)

[3.2 文件、包 8](#_Toc312056017)

[3.3 类 9](#_Toc312056018)

[3.4 接口 9](#_Toc312056019)

[3.5 常量 10](#_Toc312056020)

[3.6 变量 10](#_Toc312056021)

[3.7 方法 11](#_Toc312056022)

[3.8 集合 12](#_Toc312056023)

[3.9 异常 12](#_Toc312056024)

[3.10 界面元素的命名 13](#_Toc312056025)

[3.11 其他 13](#_Toc312056026)

[第4章 注释 14](#_Toc312056027)

[4.1 基本原则 14](#_Toc312056028)

[4.2 注释类型 14](#_Toc312056029)

[4.2.1 实现注释的格式 15](#_Toc312056030)

[4.2.1.1 块注释 15](#_Toc312056031)

[4.2.1.2 单行注释 15](#_Toc312056032)

[4.2.1.3 尾端注释 16](#_Toc312056033)

[4.2.1.4 行末注释 16](#_Toc312056034)

[4.2.2 文档注释（JavaDoc） 17](#_Toc312056035)

[4.3 文件与包注释 17](#_Toc312056036)

[4.4 类与接口注释 18](#_Toc312056037)

[4.5 方法注释 18](#_Toc312056038)

[4.6 其它注释 19](#_Toc312056039)

[4.7 注释参考表 19](#_Toc312056040)

[第5章 Java源文件内容规范 20](#_Toc312056041)

[5.1 开头注释 21](#_Toc312056042)

[5.2 包和引入 21](#_Toc312056043)

[5.3 类和接口声明 21](#_Toc312056044)

[第6章 声明 22](#_Toc312056045)

[6.1 每行声明变量的数量 22](#_Toc312056046)

[6.2 初始化 22](#_Toc312056047)

[6.3 布局 22](#_Toc312056048)

[6.4 包的声明 23](#_Toc312056049)

[6.5 类和接口的声明 23](#_Toc312056050)

[6.6 方法 24](#_Toc312056051)

[6.6 字段 24](#_Toc312056052)

[6.7 声明规则 25](#_Toc312056053)

[6.8 示例 25](#_Toc312056054)

[第7章 类与接口 26](#_Toc312056055)

[7.1 基本原则 26](#_Toc312056056)

[7.2 抽象类与接口 26](#_Toc312056057)

[7.3 继承与组合 27](#_Toc312056058)

[7.4 构造函数和静态工厂方法 27](#_Toc312056059)

[第8章 方法 27](#_Toc312056060)

[8.1 基本原则 27](#_Toc312056061)

[8.2 参数和返回值 28](#_Toc312056062)

[第9章 表达式与语句 28](#_Toc312056063)

[9.1 基本原则 28](#_Toc312056064)

[9.2 控制语句 29](#_Toc312056065)

[9.2.1 if语句 29](#_Toc312056066)

[9.2.2 switch语句 30](#_Toc312056067)

[9.3 循环语句 30](#_Toc312056068)

[9.3.1 do while语句 30](#_Toc312056069)

[9.3.2 while语句 31](#_Toc312056070)

[9.3.3 for语句 31](#_Toc312056071)

[9.4 转向语句 32](#_Toc312056072)

[9.5 try-catch语句 32](#_Toc312056073)

[第10章 错误与异常 33](#_Toc312056074)

[10.1 基本原则 33](#_Toc312056075)

[10.2 已检查异常与运行时异常 33](#_Toc312056076)

[10.3 异常的捕捉与处理 33](#_Toc312056077)

[第11章 性能与安全 34](#_Toc312056078)

[11.1 基本原则 34](#_Toc312056079)

[11.2 String与StringBugffer 34](#_Toc312056080)

[11.3 集合 35](#_Toc312056081)

[11.4 对象 35](#_Toc312056082)

[11.5 同步 35](#_Toc312056083)

[11.6 final 35](#_Toc312056084)

[11.7 垃圾收集和资源释放 36](#_Toc312056085)

[第12章 可移植性 36](#_Toc312056086)

[第13章 日志组件 36](#_Toc312056087)

[第14章 附录 37](#_Toc312056088)

[14.1 CVS标识符 37](#_Toc312056089)

[14.2 常用缩写简表 37](#_Toc312056090)

# 第1章 绪论

## 1.1 目的

本文编写Java编码规范的目的主要是为Java软件开发者提供一个标准的、规范的方式进行代码的设计和编写。通过遵循此代码规范，以使开发人员养成良好的编码风格和习惯，形成开发小组编码约定，最终提高程序的可靠性、可读性、可修改性、可维护性和一致性等，增进团队间的交流，从而进一步提高软件开发团队的效率。

## 1.2 范围

本文档对所有采用Java作为开发语言的项目适用。

## 1.3 参考资料

《Java开发规范》(Java Development Specification) 作者：韩卿

《Java编码规范及实践》

《Java编码规范》 作者：张玮

《Sun Java语言编码规范》(Java Code Conventions)

《Coding Style》作者：刘建勋

## 1.4 概述

本文档共分为十四部分：

第一部分为简介，介绍了本文的编写目的、文档适用范围和参考资料；

第二部分为代码组织与风格，介绍了在Java代码编写中需要注意的一些通用格式；

第三部分为命名规范，介绍了命名的基本原则和编写Java代码需要注意的命名规范；

第四部分为注释，介绍了注释的基本原则、注释的类型以及如何写注释；

第五部分为Java源文件内容规范，主要介绍了Java源文件的格式及包含的内容；

第六部分为声明，介绍了声明的规则以及如何进行声明；

第七部分为类与接口，介绍了编写类与接口的基本原则和需要注意s的方面；

第八部分为方法，介绍了方法的基本原则以及写参数和返回值时s需要注意的方面；

第九部分为表达式和语句，介绍了编写表达式和语句的基本原则，编写控制语句、循环语句、转向语句和try-catch的的格式和注意方面；

第十部分为错误与异常，介绍了基本原则，处理错误与异常的方法。

第十一部分为性能与安全，介绍了字符串、集合、对象、同步、final、垃圾收集和资源释放在性能与安全上是如何处理的。

第十二部分为可移植性，介绍了做到可移植性需要注意的地方。

第十三部分为日志组件，介绍了Log中方法的使用。

第十四部分为附录，介绍了CVS标识符和常用的缩写。

# 第2章 代码组织与风格

## 2.1 基本原则

代码组织与风格的基本原则是：便于自己的开发，易于与他人的交流。

不管是因个人习惯形成的代码组织与风格，还是由编辑器设置形成的代码组织与风格，都要求代码组织与风格必须前后一致，并符合本规范的基本要求和原则。

## 2.2 括号

在表达式、方法调用及方法声明中圆括号“(”后及“)”前不应该有空格。一般而言，在含有多种运算符的表达式中使用括号来避免运算符优先级问题，是个好方法。即便运算符的优先级对你而言可能很清楚，但对其他人未必如此。你不能假设别的程序员和你一样清楚运算符的优先级。

例如：

|  |
| --- |
| if(a == b && c ==d ) // 避免  if((a == b) && (c == d)) // 推荐 |

用大括号‘{’和‘}’界定一段程序块时，编写程序块时‘{’和 ‘}’应各独占一行，左括号"{"当紧跟其语句后，右括号"}"永远单独作为一行且与其匹配行对齐，并尽量在其后说明其匹配的功能模块。

较长的方法以及类、接口等的右括号”}”后应使用//end ...等标识其结束。如:

|  |
| --- |
| 类的结束符：}//end 类名，  方法结束符：}//end 方法名，s  功能块结束：}//end if...什么功能  循环块结束：}//end for...循环什么 |

左括号是否换行等随个人习惯而定，若换行则当与其前语句首字符对齐。

## 2.3 缩进

函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格。采用TAB 键缩进，在编辑器中请将TAB设置为以空格替代，否则在不同编辑器或设置下会导致TAB长度不等而影响整个程序代码的格式。

代码的新行与上一行根据逻辑关系要进行适应的缩进（一个tab键）。一般，同级之间在同一个缩进位置；下一级与上一级之间需要缩进。当功能块过多而导致缩进过深时，应将子功能块提取出来做为子函数。

例如：

|  |
| --- |
| public void methodName(){  if(some condition){  for(…){  //some sentences  }//end for  }//end if  } |

## 2.4 空白行

类、方法及功能块间等应以空白行相隔，以增加可读性，但不得有无规则的大片空白行。建议采用一个空白行来分隔代码的逻辑组，例如控制结构；采用两个空行来分隔不同方法的定义。

## 2.5 空格

关键词（或变量）和操作符之间加一个空格。例如：

|  |
| --- |
| int iCont = 1; //操作符和值之间有一个空格。 |

关键字与其后紧挨的括号之间应有一个空格进行分隔；参数列表的逗号之后应有一个空格进行分隔；大括号开始之前应有一个空格进行分隔；所有的二元操作符的前后均应有一个空格，二元操作符的例子有加号，除号，等于符号，赋值符号等；for的每一个语句之间应有空格，即for (expr1; expr2; expr3)；强制类型转换的括号之后应有一个空格，如：

|  |
| --- |
| myMethod((byte) aNum, (Object) x);  myMethod((int) (cp + 5), ((int) (i + 3)) + 1); |

## 2.6 长度

为便于阅读和理解，单个函数的有效代码长度当尽量控制在100行以内（不包括注释行），当一个功能模块过大时往往造成阅读困难，因此当使用子函数等将相应功能抽取出来，这也有利于提高代码的重用度。

单个类也不宜过大，当出现此类情况时当将相应功能的代码重构到其他类中，通过组合等方式来调用，建议单个类的长度包括注释行不超过1500行。

尽量避免使用大类和长方法。

## 2.7 行宽

尽量避免一行长度超过100个字符，当超过这个宽度，可能会导致在某些机器中无法以一屏来完整显示, 影响阅读，但这一设置也可以灵活调整。在任何情况下, 超长的语句应该在一个逗号后或一个操作符前折行。一条语句折行后, 应该比原来的语句再缩进一个TAB或4个空格，以便于阅读。

## 2.8 换行

当一行装不下内容时，需要对这些内容进行折行，折行时遵守以下规则：

* 在操作符后折行，操作符包括+，-，\*，/，左括号，逗号，”&&”，”||”，问号，冒号，例如：

|  |
| --- |
| someMethod(longExpression1 , longExpression2 , longExpression3 ,  longExpression4, longExpression5);    var = someMethod1(longExpression1,  someMethod2(longExpression2,  longExpression3)); |

* 选择层次较高（级别较高）的进行折行

同一层次是指在表达式树中同一级（同一个括号内），层次较高是指在表达式树中较高的一级，以下两个例子可以说明这个规则。

|  |
| --- |
| longName1 = longName2 \* (longName3 + longName4 - longName5)  + 4 \* longname6; // 建议  longName1 = longName2 \* (longName3 + longName4  - longName5) + 4 \* longname6; // 避免 |

* 方法定义的折行

如果方法定义太长，可以在参数处进行折行。

以下是两个例子，前者是常规情形，后者若使用常规的缩进方式将会使第二行和第三行移得很靠右，所以代这以缩进8个空格。如：

|  |
| --- |
| //同级开始处对齐原则  someMethod(int anArg, Object anotherArg, String yetAnotherArg,  Object andStillAnother) {  ...  }  //比上一行缩进一级原则  private static synchronized horkingLongMethodName(int anArg,  Object anotherArg, String yetAnotherArg,  Object andStillAnother) {  ...  } |

* if语句的换行通常使用8个空格的规则，因为常规缩进(4个空格)会使语句看起来比较费劲。比如：

|  |
| --- |
| //避免  if ((condition1 && condition2)  || (condition3 && condition4)  || !(condition5 && condition6)) { //不好  doSomethingAboutIt(); // 导致此行代码与上一行不好区分  }  //建议使用这样的方式  if ((condition1 && condition2)  || (condition3 && condition4)  || !(condition5 && condition6)) {  doSomethingAboutIt();  }  //或是这样的方式  if ((condition1 && condition2) || (condition3 && condition4)  || !(condition5 && condition6)) {  doSomthingAoutIt();  } |

* 这里有三种可行的方法用于处理三元运算表达式：

|  |
| --- |
| alpha = (aLongBooleanExpression) ? beta : gamma;  alpha = (aLongBooleanExpression) ? beta  : gamma;  alpha = (aLongBooleanExpression)  ? beta  : gamma; |

## 2.9 对齐

* 关系密切的行应对齐，对齐包括类型、修饰、名称、参数等各部分对齐。
* 连续赋值时当对齐操作符。
* 当方法参数过多时当在每个参数后（逗号后）换行并对齐。
* 当控制或循环中的条件比较长时当换行（操作符前）、对齐并注释各条件。
* 变量定义最好通过添加空格形成对齐，同一类型的变量应放在一起。

如下例所示：

|  |
| --- |
| //变量对齐-----------------------------------------------  int count = 100;  int length = 0;  String strUserName = null;  Integer[] porductCode = new Integer(2); //产品编码数组  //参数对齐----------------------------------------------  public Connection getConnection(String url,  String userName,  String password)  throws SQLException,IOException{  }  //换行对齐----------------------------------------------  public final static String SQL\_SELECT\_PRODUCT = “SELECT \* “  + “ FROM TProduct WHERE Prod\_ID = ”  + prodID;  //条件对齐----------------------------------------------  if( Condition1 //当条件一  && Condition2 //并且条件二  || Condition3){ //或者条件三  } |

# 第3章 命名规范

## 3.1 基本原则

规范的命名能使程序更易阅读，从而更易于理解。它们也可以提供一些标识功能方面的信息，有助于更好的理解代码和应用。

* 使用可以准确说明变量/字段/类/接口/包等的完整的英文描述符

例如，采用类似 firstName，listAllUsers 或 CorporateCustomer 这样的名字。虽然象 x1，y1 或 fn 这样的名字很简短，输入起来容易，但是我们难以知道它们代表什么、结果是什么含义，因而使代码难以理解、维护和改进。严禁使用汉语拼音及不相关单词命名。

* 采用该领域的术语

如果用户称他们的“客户” (clients) 为“顾客” (customers)，那么就采用术语 Customer 来命名这个类，而不用 Client。许多程序开发者会犯的一个错误是，不去使用工业或领域里已经存在着很完美的术语时，却生造出一些普通词汇。

* 大小写

常量的字母全部大写，单词之间用一个下划线字符(\_)进行分隔；除常量外的命名采用大小写混合，提高名字的可读性。一般采用小写字母，但是类和接口的名字的首字母，以及任何中间单词的首字母应该大写；包名全部小写。

* 下划线的使用

避免使用下划线作为名字的首末字母，以下划线为首末字母的名字通常为系统保留，除预处理定义之外，一般不用作用户命名。更重要的是，下划线经常造成麻烦而且难输入，所以尽量避免使用。因此建议下划线只用在常量的命名的单词中间作分隔符。

* 缩写

尽量少用缩写，但如果一定要使用，当使用公共缩写和习惯缩写等，如实现（implement）可缩写成impl，经理（manager）可缩写成mgr等，具体参看附录之《常用缩写简表》，严禁滥用缩写。

* 避免使用长名字（最好不超过 25 个字母）
* 避免使用相似或者仅在大小写上有区别的名字
* 避免使用数字，但可用2代替to，用4代替for等，如：go2Jsp。

## 3.2 文件、包

* 文件名当与其类严格相同，所有单词首字母大写。
* 包名一般以项目或模块名命名，少用缩写和长名，一律小写，不要出现下划线等符号，名词用有意义的缩写或者英文单词。

|  |
| --- |
| //推荐  com.esse.business  java.lang.util  //避免  com.Esse-tech.buSiness |

* 包名按如下规则组成： [基本包].[项目名].[模块名].[子模块名]... 如：

|  |
| --- |
| org.skyinn.quasar  org.skyinn.skyhome.dao.hibernate |

每一个包的命名规则采用类似于TCP/IP中域名的反序，即以com, edu, gov, mil, net, org, 或ISO 3166,1981中定义的两位国家代码开始，接着是组织名称，在此处以com.run开始，接着是<项目英文简写>，其后的部分由项目自已定义。例如：

|  |
| --- |
| com.run.ppo.users.dao  com.run.ppo.people.action |

* 不得将类直接定义在基本包下，所有项目中的类、接口等都当定义在各自的项目和模块包中。

## 3.3 类

* 类名称内容为名词或者名词性语句，不要过长，类名要简洁而富于描述。例如：UserManager，DatabaseOperator
* 使用完整单词，避免缩写词(除非该缩写词被广泛的使用，如URL，HTML)，如果使用缩写词，要使用通用的计算机缩写或通用的领域缩写
* 组成类名称的第一个单词的首字母要大写，单词的其余部分小写或者大小写组合，例如：GetDataAction、GetDataForm
* 如果有继承或者实现关系的两个类，名称上应该能很容易区分。例：Interface：FetchDataDao implementClass: FetchDataDaoImpl

## 3.4 接口

* 同3.3节类命名规则
* 接口可带I前缀，例如：IQuery,IDataAccess,IReportBuilder
* 接口声明了一个对象能提供的服务，也描述了一个对象的能力，所以可以以able、ible、er等作为后缀，代表了一种能力。例如：

|  |
| --- |
| public interface Runnable{  public void run();  }  public interface Accessible{  public Context getContext();  } |

* 缺省接口实现应该使用Default 前缀

|  |
| --- |
| class DefaultTableCellRenderer implements TableCellRenderer{  …  } |

## 3.5 常量

* 采用完整的英文大写单词，词与词之间用下划线连接，如：

|  |
| --- |
| static final int MIN\_WIDTH = 4;  static final int MAX\_WIDTH = 999;  static final int GET\_THE\_CPU = 1;  static final String DB\_CONFIG\_FILE\_PATH = "com.neu.etrain.dbconfig"; |

* 常量名称尽量表述清楚，一般不是太长的话不用简写，如：MARGIN\_BOTTOM
* 程序中应该使用常量代替“25”，“100”等实际的数字，如：

|  |
| --- |
| //推荐  static final int MAX\_TIMES = 25;  if(times == MAX\_TIMES){  …  }  //避免  if(times == 25){  …  } |

这样做的好处是，当因需要修改实际的数字时，比如修改25，只需要修改一处，及修改static final int MAX\_TIMES = 25即可。

## 3.6 变量

* 变量的名字必须用一个小写字母开头，后面的单词首字母大写，不需要数据类型前缀，例如：

|  |
| --- |
| String userName  int size  float myWidth |

* 变量名不应以下划线或美元符号开头，尽管这在语法上是允许的
* 变量名应该短、有意义、能清晰表达出意思并便于记忆
* 尽量避免单个字符的变量名，除非是一个临时变量名。临时变量通常被取名为i，j，k，m和n，它们一般用于整形；c，d，e，它们一般用于字符型。
* 对不易清楚识别出该变量类型的变量应使用类型缩写作其前缀，如字符串使用strXXX,boolean使用isXXX或是hasXXX等等
* 变量的名字应该和类型名称一致，例如：

|  |
| --- |
| 例1：  //建议  void setTopic(Topic topic)  // 避免:  void setTopic(Topic value)  // 避免:  void setTopic(Topic aTopic)  // 避免:  void setTopic(Topic t)  例2：  //建议  void connect(Database database)  // 避免:  void connect(Database db)  // 避免:  void connect(Database oracleDB) |

* 当同时定义多个属于同一个类的变量时，把类型作为实例的后缀，如：

|  |
| --- |
| Point startPoint;  Point centerPoint; |

这样做是为了从实例名就可以推断它的类型名称。

## 3.7 方法

* 方法名应该使用动词开头，一般由动词+名词组成，例如：getName, initialize, addParameter, deleteUser
* 采用大小写混合方式，第一个单词的首字母小写，其后单词的首字母大写，例如：setName() , getMethod()
* 对变量的取得和设定一般用set+变量名称和get+变量名称，例如：getName()，setName()
* 对象的boolean条件判断，方法名称是is+对象名称或是has+对象名称，例如：isLogon()
* 对象转换特定形式一般用to+特定形式。 例如：toString()
* 方法中的参数建议顺序：(被操作者，操作内容，操作标志，其他⋯)

|  |
| --- |
| public void replace(String sourceStr, //源字串  String oldStr, //被替换字串  String newStr){ //替换为字串  } |

* 方法声明中参数命名：使用有意义的参数命名，如果可能的话，使用和要赋值的字段一样的名字。例如：

|  |
| --- |
| public void setCounter(int size){  this.size = size;  } |

## 3.8 集合

* 一个集合，例如数组和矢量，应采用复数命名来表示队列中存放的对象类型，例如：String[] userNames;
* 命名应采用完整的英文描述符，采用大小写混合方式，第一个单词的首字母小写，其后单词的首字母大写，，适当使用集合缩写前缀,如：

|  |
| --- |
| Vector vProducts = new Vector(); //产品向量  Array aryUsers = new Array(); //用户列表 |

* 数组应该总是用下面的方式来命名：

|  |
| --- |
| //建议  byte[] buffer;  //避免  byte buffer[]； |

## 3.9 异常

* 异常类名由表示该异常类型的单词和Exception组成，如ActionException，AccessException，RuntimeException。
* 异常实例一般使用e、ex等，在多个异常时使用该异常名或简写加E，Ex等组成，如： Exception e，Event e，SQLEx ActionEx

## 3.10 界面元素的命名

界面元素是指如Form，控件等变量及类的命名。界面元素命名的一条基本原则是界面元素的命名应简单而不产生混淆，一般情况下，界面元素的命名直接参照变量命名规则，但若发生了混淆或估计会发生混淆的界面元素，则可以在该元素的后加元素类型的后缀，后缀的生成规则是：

* 若元素类型是标准的Swing类型，则去掉类名”J”字符后的部分为元素类型名
* 若元素类型是标准的AWT类型，则直接使用类名
* 如果元素类型不是标准的Swing或AWT类型，元素类型的后缀使用其Swing或AWT的基类进行命名。如：自定义了class MyPanel extends JPanel，则若有MyPanel的元素，则元素后缀为Panel。

下表给出了基本的元素类型后缀：

|  |  |  |
| --- | --- | --- |
| **类型** | **后缀** | **例** |
| JPanel | Panel | listPanel |
| JButton | Button | delButton |
| JFrame | Frame | mainFrame |
| JInternalFrame | Frame或InternalFrame | welcomeFrame,welcomeInternalFrame |
| JCheckBox | CheckBox |  |

## 3.11 其他

* 在查询方法中应使用find 作为前缀，例如：findSmallestElement()，findShortestPath(Node destinationNode)
* 使用initialize 做为对象初始化的方法前缀，也可以简写为init，例如：initializeFiles()，init()，initFontSet()
* 对于抽象类，应该使用Abstract 前缀，例如：AbstractReportBuilder，AbstractBeanFactory
* 对于表示编号的变量，应加No 后缀，例如：tableNo, userNo,employeeNo
* 建议使用的对称词汇，这些词汇一起使用，方法的表达意图自然可以互相推测和演绎。例如：

|  |
| --- |
| get/set, add/remove, create/destroy,start/stop,  insert/delete, increment/decrement, begin/end, first/last,  up/down, min/max, next/previous, old/new, open/close,  show/hide, suspend/resume |

* 避免使用否定布尔变量

|  |
| --- |
| bool isError; *//* 避免*: isNoError*  bool isFound; *//* 避免*: isNotFound* |

* 对于工厂类，进行创建对象的方法，应该使用new 前缀

|  |
| --- |
| class PointFactory {  public Point newPoint(...) {  ...  }  } |

# 第4章 注释

## 4.1 基本原则

* 在必要的地方注释，注释量要适中
* 注释的内容要清楚、明了，含义准确，防止注释含义不明确。
* 保持注释与其描述的代码相邻，即注释的就近原则
* 边写代码边注释，修改代码同时修改相应的注释，以保证注释与代码的一致性
* 注释信息不仅要包括代码的功能，还应给出原因
* 除变量定义等较短语句的注释可用行尾注释外，其他注释当避免使用行尾注释。

## 4.2 注释类型

Java程序有两类注释：实现注释(implementation comments)和文档注释(document comments)。实现注释是那些在C++中见过的，使用/\*…\*/和//界定的注释。文档注释(被称为“doc comments”)是Java独有的，并由/\*\*…\*/界定。文档注释可以通过javadoc工具转换成HTML文件。

实现注释用以注释代码或者实现细节。文档注释从实现自由(implemtentation-free)的角度描述代码的规范。它可以被那些手头没有源码的开发人员读懂。

注释不应写在用星号或字符画出来的大框里。注释不应包括诸如制表符和回退符之类 的特殊字符。

注意：频繁的注释有时反映出代码的低质量。当你觉得被迫要加注释的时候，考虑一下重写代码使其更清晰。

## 4.2.1 实现注释的格式

程序可以有4种实现注释的风格：块(Block)，单行(single-line)，尾端(trailing)和行末(end-of-line)。

## 4.2.1.1 块注释

块注释通常用于提供对文件，方法，数据结构和算法的描述。块注释被置于每个文件的开始处以及每个方法之前。它们也可以被用于其他地方，比如方法的内部。在功能和方法内部的块注释应该和它们所描述的代码具有一样的缩进格式。

块注释之首应该有一个空行，用于把块注释和代码分割开来，比如：

|  |
| --- |
| /\*  \* 块注释  \*/ |

## 4.2.1.2 单行注释

单行注释主要用在方法内部，对代码，变量，流程等进行说明，并与其后的代码具有一样的缩进层级。单行注释只占据一行，如果一个注释不能在一行内写完，就该块注释(参见“块注释”)。单行注释之前应该有一个空行。以下是一个Java代码中单行注释的例子：

|  |
| --- |
| if (condition) {    /\* 单行注释 \*/  ……  } |

## 4.2.1.3 尾端注释

极短的注释可以与它们所要描述的代码位于同一行，但是应该有足够的空白来分开代码和注释。若有多个短注释出现于大段代码中，它们应该具有相同的缩进。

以下是一个Java代码中尾端注释的例子：

|  |
| --- |
| if (a ==2) {  return TRUE; / \* special case \*/  } else {  return isPrime(a); / \* works only for odd a \*/  } |

## 4.2.1.4 行末注释

注释界定符“//”，可以注释掉整行或者一行中的一部分。它一般不用于连续多行的注释文本；然而，它可以用来注释掉多行的代码段。以下是所有三种风格的例子：

|  |
| --- |
| If (foo > 1) {  // Do a double-filp.  ……  } else {  return false; // Explain why here.  }  // if (bar > 1) {  //  // // Do a triple-filp.  // ……  // } else {  // return false;  // } |

## 4.2.2 文档注释（JavaDoc）

文档注释描述Java的类、接口、构造器、方法，以及字段(field)。每个文档注释都会被置于注释界定符/\*\*…\*/之中，一个注释对应一个类、接口或成员。该注释应位于声明之前, 文档注释不能放在方法或程序块内。例如：

|  |
| --- |
| /\*\*  Customer是指作为我们的服务及产品的销售对象的任何个人或组织。  @author S.W. Ambler  \*/  public class Customer { … |

javadoc注释标签语法:

|  |
| --- |
| @author 对类的说明 标明开发该类模块的作者  @version 对类的说明 标明该类模块的版本  @see 对类、属性、方法的说明 参考转向，也就是相关主题  @param 对方法的说明 对方法中某参数的说明  @return 对方法的说明 对方法返回值的说明  @exception 对方法的说明 对方法可能抛出的异常进行说明 |

## 4.3 文件与包注释

在每个文件、包的头部都应该包含该文件的功能、作用、作者、版权以及创建、修改记录等。并在其中使用CVS标记(见附录)自动跟踪版本变化及修改记录等信息。注意是/\*\*/注释而不是/\*\*\*/JavaDoc注释。

文件注释示例：

|  |
| --- |
| /\* =========================================================  \* $Id: User.java,v 1.1 2002/09/07 14:36:23 l\_walker Exp $  \* Created: [2011-12-16 20:18:53] by BrendaD  \* =========================================================  \*  \* ProjectName  \*  \* Description  \*  \* =========================================================  \*  \* Copyright Information.  \*  \* Modify BrendaD (如果有多个，依次写多个modify)  \* =========================================================  \*/ |

每个包当有包注释，在源码及JavaDoc相应包路径下建立package.html以描述包的功能、作用等。

## 4.4 类与接口注释

在类、接口定义之前当对其进行注释，包括类、接口的目的、作用、功能、继承于何种父类，实现的接口、实现的算法、使用方法、示例程序等，在作者和版本域中使用CVS标记自动跟踪版本变化等。

类注释示例：

|  |
| --- |
| /\*\*  \* <p>字符串实用类。</p>  \*  \* 定义字符串操作时所需要用到的方法，如转换中文、HTML标记处理等。  \*  \* @author $Author: l\_walker$  \* @version $Revision: 1.2 $ $Date: 2003/05/15 02:10:27 $  \*/  public class StringUtil {  ⋯  } |

## 4.5 方法注释

依据标准JavaDoc规范对方法进行注释，以明确该方法功能、作用、各参数含义以及返回值等。复杂的算法用/\*\*/在方法内注解出。

* 参数注释时当注明其取值范围等
* 返回值当注释出失败、错误、异常时的返回情况。
* 异常当注释出什么情况、什么时候、什么条件下会引发什么样的异常

方法注释示例：

|  |
| --- |
| /\*\*  \* 执行查询。  \*  \* 该方法调用Statement的executeQuery(sql)方法并返回ResultSet  \* 结果集。  \*  \* @param sql 标准的SQL语句  \* @return ResultSet结果集，若查询失败则返回null  \* @throws SQLException 当查询数据库时可能引发此异常  \*/  public ResultSet executeQuery(String sql) throws SQLException  {  //Statement和SQL语句都不能为空  if(null != stmt && !StringUtil.isEmpty(sql)){  //返回查询执行结果  return stmt.executeQuery(sql);  }  return null;  }//end executeQuery() |

## 4.6 其它注释

* 应对重要的变量、常量加以注释，以说明其含义等，注释应放在其上方相邻位置或右方，不可放在下面。
* 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。
* 应对不易理解的分支条件表达式加注释。
* 不易理解的循环，应说明出口条件。
* 过长的方法实现，应将其语句按实现的功能分段加以概括性说明。
* 对于异常处理当注明正常情况及异常情况或者条件，并说明当异常发生时程序当如何处理。

## 4.7 注释参考表

|  |  |
| --- | --- |
| 项目 | 注释内容 |
| 参数 | 参数类型  参数用来做什么  约束或前提条件  示例 |
| 字段/属性 | 字段描述  注释所有使用的不变量  示例  并行事件  可见性决策 |
| 类 | 类的目的  已知的问题  类的开发/维护历史、版本  注释出采用的不变量  并行策略 |
| 编译单元  （文件） | 每一个类/类内定义的接口，含简单的说明  文件名和/或标识信息  修改/维护记录  版权信息 |
| 获取成员函数 | 若可能，说明为什么使用滞后初始化 |
| 接口 | 目的  它应如何被使用以及如何不被使用 |
| 局部变量 | 用处/目的 |
| 成员函数注释 | 成员函数做什么以及它为什么做这个  哪些参数必须传递给一个成员函数  成员函数返回什么  已知的问题  任何由某个成员函数抛出的异常  可见性决策  成员函数是如何改变对象的  包含任何修改代码的历史  如何在适当情况下调用成员函数的例子  适用的前提条件和 后置条件 |
| 成员函数内部注释 | 控制结构  代码做了些什么以及为什么这样做  局部变量 |

# 第5章 Java源文件内容规范

每个Java源文件都包含一个单一的公共类或接口。若私有类和接口与一个公共类相关联，可以将它们和公共类放入同个源文件。公共类必须是这个文件中的第一个类和接口。

Java源文件还遵循以下规则：

* 开头注释
* 包和引入语句
* 类和接口声明

## 5.1 开头注释

所有的源文件都应该在开头有一个C语言风格的注释，其中列出类名、版本信息，日期和版权声明，变更记录包含变更日期、变更人和变更内容：

|  |
| --- |
| /\*  \* Classname  \*  \* Version information  \*  \* Date  \*  \* Copyright notice  \*/ |

## 5.2 包和引入

在多数Java源文件中，第一个非注释行是包语句行。在它之后可以跟引入语句。例如：

|  |
| --- |
| package java.awt;  import java.awt.peer.CanvasPeer; |

## 5.3 类和接口声明

下表描述了类和接口声明的免修部分以及它们出现的先后次序。

|  |  |  |
| --- | --- | --- |
|  | 类/接口声明的各部分 | 注解 |
| 1 | 类/接口文档注释  (/ \*\*…\* /) | 该注释中所包含的信息 |
| 2 | 类/接口的声明 |  |
| 3 | 类/接口实现的注释  (/ \*…\* /)  如果有必要的话 | 该注释应包含任何有关整个类或接口的信息，而这些信息又适合作为类/接口文档注释。 |
| 4 | 类的(静态)变量 | 首先是类的public变量，随后是protected变量，再后是包一级别的变量(没有访问修饰符)，最后是private变量。 |
| 5 | 实例变量 | 首先是public变量，随后是protected变量，再后是包一级别的变量(没有访问修饰符)，最后是private变量。 |
| 6 | 构造器 |  |
| 7 | 方法 | 这些方法应该按功能，而非作用域或访问权限，分组。 |

# 第6章 声明

## 6.1 每行声明变量的数量

每行应该只有一个声明，因为这样以利于写注释。例如：

|  |
| --- |
| //推荐  int level; // indentation level  int size; // size of table  //避免  int level, size; |

## 6.2 初始化

局部变量必须初始化

## 6.3 布局

* 只在代码块的开始处声明变量。（一个块是指任何被包含在大括号“{”和“}”中间的代码）

|  |
| --- |
| void myMethod() {  int int1 = 0; // 方法块的开始  if (condition) {  int int2 = 0; // if代码块的开始  ...  }  } |

该规则的一个例外是for循环的索引变量，例如：

|  |
| --- |
| for (int i = 0; i < maxLoops; i++) {  ...  } |

* 不要在首次用到该变量时才声明之。这会把注意力不集中的程序员搞糊涂，同时会妨碍代码在该作用域内的可移植性。
* 避免声明的局部变量覆盖上一级声明的变量。例如，不要在内部代码块中声明相同的变量名：

|  |
| --- |
| int count;  ...  myMethod() {  if (condition) {  int count = 0; //避免  ...  }  ...  } |

## 6.4 包的声明

在导入包时，应当完全限制代码所使用的类的名字,尽量少用通配符的方式，但导入一些通用包，或用到一个包下大部分类时，则可是使用通配符方式，如：

|  |
| --- |
| import org.skyinn.quasar.services.Service;  import java.util.\*; |

## 6.5 类和接口的声明

类、接口定义语法规范：

[可见性][('abstract'|'final')] [Class|Interface] class\_name

[('extends'|'implements')][父类或接口名]{

}

如：

|  |
| --- |
| public class LoginAction extends BaseAction implemnets ActionListener {  ……  } |

当编写类和接口是，应该遵守以下格式规则：

* 在方法名与其参数列表之前的左括号"("间不要有空格；
* 左大括号“{”位于声明语句同行的末尾；
* 右大括号“}”另起一行，与相应的声明语句对齐，除非是一个空语句，“}”应紧跟在“{”之后；
* 方法与方法之间以空行分隔。

|  |
| --- |
| class Sample extends Object {  int ivar1;  int ivar2;  Sample(int i, int j) {  ivar1 = i;  ivar2 = j;  }  int emptyMethod() {}  String myMethod() {  ...  }  ...  } |

## 6.6 方法

良好的程序设计应该尽可能减小类与类之间耦合，所遵循的经验法则是：尽量限制成员函数的可见性。如果成员函数没必要公有 (public)，就定义为保护 (protected)；没必要保护 (protected)，就定义为私有 (private)。

方法的定义语法规范：

[可见性][('abstract'|'final')] ['synchronized'][返回值类型] method\_name(参数列表)[('throws')][异常列表]{

// 功能模块

}

如：

|  |
| --- |
| public List listAllUsers() throws DAOException{  ...  } |

## 6.6 字段

字段的定义语法规范：

[(‘public’|’private’|’protected’)]

[(‘final’|’volatile’)][‘static’][‘transient’]

data\_type field\_name [ ‘=’ expression] ‘;’

若没有足够理由，不要把实例或类变量声明为公有。公共和保护的可见性应当尽量避免，所有的字段都建议置为私有，由获取和设置成员函数（Getter、Setter）访问。

不允许“隐藏”字段，即给局部变量所取的名字，不可与另一个更大范围内定义的字段的名字相同（或相似）。例如，如果把一个字段叫做 firstName ，就不要再生成一个局部变量叫做 firstName，或者任何易混肴的名字。

## 6.7 声明规则

* 修饰符应该按照如下顺序排列：public, protected, 默认, private, abstract, static, final, transient, volatile, synchronized, native, strictfp。
* 字段的声明顺序：常量，类变量，实例变量，公有字段，受保护字段，私有字段
* 方法的声明顺序： 构造方法，静态公共方法，静态私有方法，受保护方法，私有方法，继承自Object的方法

## 6.8 示例

|  |
| --- |
| //常量---------------------------------------------------  public final static double PI = 3.141592653589793;  // -------------------------------------------------类变量  protected static String key = “Love”;  // -----------------------------------------------实例变量  //公有字段-----------------------------------------------  public String userName = “Tom”;  //受保护字段---------------------------------------------  protected float price = 0.0;  //私有字段-----------------------------------------------  private int count;  //构造方法-----------------------------------------------  public Constructor(){  }  //公共方法-----------------------------------------------  public String getUserName(){  }  //友元方法-----------------------------------------------  void createProduct(){  }  //受保护方法---------------------------------------------  protected void convert(){  }  //私有方法-----------------------------------------------  private void init(){  }  //重载Object方法----------------------------------------  public String toString(){  }  //main方法-----------------------------------------------  public static void main(String[] args){  } |

# 第7章 类与接口

## 7.1 基本原则

* 类的划分粒度，不可太大，造成过于庞大的单个类，也不可太细，从而使类的继承太深。一般而言，一个类只做一件事；另一个原则是根据每个类的职责进行划分，比如用User来存放用户信息，而用UserDAO来对用户信息进行数据访问操作（比如存取数据库）等等。
* 多使用设计模式，随时重构。
* 多个类中使用相同方法时将其方法提到一个接口中或使用抽象类，尽量提高重用度。
* 将不希望再被继承的类声明成final，例如某些实用类，但不要滥用final，否则会对系统的可扩展性造成影响。
* 将不希望被实例化的类的缺省构造方法声明成private。

## 7.2 抽象类与接口

一般而言：接口定义行为，而抽象类定义属性和公有行为，注意两者间的取舍，在设计中，可由接口定义公用的行为，由一个抽象类来实现其部分或全部方法，以给子类提供统一的行为定义，可参考Java集合等实现。

多使用接口，尽量做到面向接口的设计，以提高系统的可扩展性。

## 7.3 继承与组合

尽量使用组合来代替继承，一则可以使类的层次不至于过深，而且会使类与类，包与包之间的耦合度更小，更具可扩展性。

## 7.4 构造函数和静态工厂方法

当需要使用多个构造函数创建类时，建议使用静态工厂方法替代这些构造方法，例如：

|  |
| --- |
| public class User{  public User(){  super();  //do somethings to create user instance  }  public static User getInstance(String name,String password){  User u = new User();  u.setName(name);  u.setPassword(password);  return u;  } |

# 第8章 方法

## 8.1 基本原则

* 一个方法只完成一项功能，在定义系统的公用接口方法外的方法应尽可能的缩小其可见性。
* 避免用一个类是实例去访问其静态变量和方法。
* 避免在一个较长的方法里提供多个出口：

|  |
| --- |
| //不要使用这钟方式，当处理程序段很长时将很难找到出口点  if(condition){  return A;  }else{  return B;  }  //建议使用如下方式  String result = null;  if(condition){  result = A;  }else{  result = B;  }  return result; |

## 8.2 参数和返回值

* 避免过多的参数列表，尽量控制在5个以内，若需要传递多个参数时，当使用一个容纳这些参数的对象进行传递，以提高程序的可读性和可扩展性。
* 参数类型和返回值尽量接口化，以屏蔽具体的实现细节，提高系统的可扩展性，例如：

|  |
| --- |
| public void joinGroup(List userList){}  public List listAllUsers(){} |

# 第9章 表达式与语句

## 9.1 基本原则

* 表达式和语句当清晰、简洁，易于阅读和理解，避免使用晦涩难懂的语句。
* 每行至多包含一条执行语句，过长当换行。
* 避免在构造方法中执行大量耗时的初始化工作，应当将这种工作延迟到被使用时再创建相应资源，如果不可避免，则当使用对象池和Cache等技术提高系统性能。
* 避免在一个语句中给多个变量赋相同的值。
* 不要使用内嵌(embedded)赋值运算符试图提高运行时的效率，这是编译器的工作。
* 尽量在声明局部变量的同时初始化。唯一不这么做的理由是变量的初始值依赖于某些先前发生的计算。
* 一般而言，在含有多种运算符的表达式中使用圆括号来避免运算符优先级问题，是个好方法。
* 不要为了表现编程技巧而过分使用技巧，简单就好。

## 9.2 控制语句

## 9.2.1 if语句

* 判断中若有常量，则应将常量置与判断式的右侧。如：

|  |
| --- |
| if (true == isAdmin())...  if (null == user)... |

* 尽量不使用三目条件判断。
* 所有if语句必须用{}包括起来,即便是只有一句：

|  |
| --- |
| //建议  if (true){  //do something......  }  //避免  if (true)  i = 0; //不要使用这种 |

* 当有多个else分句时当分别注明其条件，注意缩进并对齐，如：

|  |
| --- |
| //先判断i 是否等于1  if (i == 1){//if\_1  //.....  } else if (i == 2){ //然后判断i == 2  //i == 2说明。。。。。。  j = i;  } else{ //如果都不是(i > 2 || i < 1)  //说明出错了  //....  }//end if\_1 |

* 过多的else分句请将其转成switch语句或使用子函数。

## 9.2.2 switch语句

一个switch语句应该具有如下格式：

|  |
| --- |
| switch (condition) {  case ABC:  statements;  /\* falls through \*/  case DEF:  statements;  break;  case XYZ:  statements;  break;  default:  statements;  break;  } |

每当一个case顺着往下执行时(因为没有break语句)，通常应在break语句的位置添加注释。上面的示例代码中就包含注释/\* falls through \*/。

## 9.3 循环语句

循环中必须有终止循环的条件或语句，避免死循环。

## 9.3.1 do while语句

一个do-while语句应该具有如下格式：

|  |
| --- |
| do {  statements;  } while (condition); |

## 9.3.2 while语句

一个while语句应该具有如下格式：

|  |
| --- |
| while (condition) {  statements;  } |

一个空的while语句应该具有如下格式：

|  |
| --- |
| while (condition); |

## 9.3.3 for语句

一个for语句应该具有如下格式：

|  |
| --- |
| for (initialization; condition; update) {  statements;  } |

一个空的for语句应该具有如下格式：

|  |
| --- |
| for (initialization; condition; update); |

当在for语句的初始化或更新子句中使用逗号时，避免因使用三个以上变量，而导致复杂度提高。若需要，可以在for循环之前(为初始化子句)或for循环末尾(为更新子句)使用单独的语句。

因为循环条件在每次循环中多会执行一次，故尽量避免在其中调用耗时或费资源的操作，比较一下两种循环的差异：

|  |
| --- |
| //不推荐方式\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  while(index < products.getCount()){  //每此都会执行一次getCount()方法，  //若此方法耗时则会影响执行效率  //而且可能带来同步问题，若有同步需求，请使用同步块或同步方法  }  //推荐方式\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  //将操作结构保存在临时变量里，减少方法调用次数  final int count = products.getCount();  while(index < count){  } |

## 9.4 转向语句

一个带返回值的return语句不使用小括号“()”，除非它们以某种方式使返回值更为显见。例如：

|  |
| --- |
| return;  return myDisk.size();  return (size ? size : defaultSize); |

## 9.5 try-catch语句

一个try-catch语句应该具有如下格式：

|  |
| --- |
| try {  statements;  } catch (ExceptionClass e) {  statements;  } |

一个try-catch语句后面也可能跟着一个finally语句，不论try代码块是否顺利执行完，它都会被执行。

|  |
| --- |
| try {  statements;  } catch (ExceptionClass e) {  statements;  } finally {  statements;  } |

# 第10章 错误与异常

## 10.1 基本原则

* 通常的思想是只对错误采用异常处理：逻辑和编程错误，设置错误，被破坏的数据，资源耗尽，等等。
* 通常的法则是系统在正常状态下以及无重载和硬件失效状态下，不应产生任何异常。
* 异常处理时可以采用适当的日志机制来报告异常，包括异常发生的时刻。
* 不要使用异常实现控制结构。
* 若有finally子句，则不要在try块中直接返回，亦不要在finally中直接返回。

## 10.2 已检查异常与运行时异常

已检查异常必须捕捉并做相应处理，不能将已检查异常抛到系统之外去处理。

捕捉并处理可预见的运行时异常，比如空指针等。通常，对空指针的判断不是使用捕捉NullPointException的方式，而是在调用该对象之前使用判断语句进行直接判断，如：

|  |
| --- |
| //若不对list是否为null进行检查，则在其为null时会抛出空指针异常  if(null != list && 0 < list.size()){  for(int i = 0; i < list.size(); i++){  }  } |

建议使用运行时异常(RuntimeException)代替已检查异常（CheckedException）.

## 10.3 异常的捕捉与处理

捕捉异常是为了处理它，不要捕捉了却什么都不处理就将其抛弃，最低限度的向控制台输出当前异常，如果你不想处理它，请将该异常抛给它的调用者，建议对每个捕捉到的异常都调用printStackTrace()输出异常信息，避免异常的湮没。

多个异常应分别捕捉并处理，避免使用一个单一的catch来处理。如：

|  |
| --- |
| try {  //do something  }catch(IllegalStateException IllEx){  IllEx.printStackTrace();  //deal with IllEx  }catch(SQLException SQLEx){  SQLEx.printStackTrace();  throw SQLEx; //抛给调用者处理  }finally{  //释放资源  } |

# 第11章 性能与安全

## 11.1 基本原则

性能的提升并不是一蹴而就的，而是由良好的编程积累的，虽然任何良好的习惯和经验所提升的性能都十分有限，甚至微乎其微，但良好的系统性能却是由这些习惯积累而成，不积细流，无以成江海！

## 11.2 String与StringBugffer

* 不要使用如下String初始化方法：

|  |
| --- |
| String str = new String(“abcdef”); |

上述方法将产生两个对象，应当直接赋值，如下：

|  |
| --- |
| String str = “abcdef”; |

* 在处理可变 String 的时候要尽量使用 StringBuffer 类。

当我们在构造字符串的时候，我们应该用 StringBuffer 来实现大部分的工作，当工作完成后将 StringBuffer 对象再转换为需要的 String 对象。比如：如果有一个字符串必须不断地在其后添加许多字符来完成构造，那么我们应该使用 StringBuffer 对象和她的 append() 方法。如果我们用 String 对象代替 StringBuffer 对象的话，将会花费许多不必要的创建和释放对象的 CPU 时间。

## 11.3 集合

避免使用Vector和HashTable等旧的集合实现，这些实现的存在仅是为了与旧的系统兼容，而且由于这些实现是同步的，故而在大量操作时会带来不必要的性能损失。在新的系统设计中不当出现这些实现，使用ArrayList代替Vector，使用HashMap代替HashTable。

若却是需要使用同步集合类，当使用如下方式获得同步集合实例：

|  |
| --- |
| Map map = Collections.synchronizedMap(new HashMap()); |

由于数组、ArrayList与Vector之间的性能差异巨大（具体参见《Java fitball》），故在能使用数组时不要使用ArrayList，尽量避免使用Vector。

## 11.4 对象

* 避免在循环中频繁构建和释放对象。
* 不再使用的对象应及时销毁。
* 如无必要，不要序列化对象。

## 11.5 同步

* 在不需要同步操作时避免使用同步操作类。
* 尽量少用同步方法，避免使用太多的 synchronized 关键字。
* 尽量将同步最小化，即将同步作用到最需要的地方，避免大块的同步块或方法等。

## 11.6 final

* 将参数或方法声明成final可提高程序响应效率

注意:绝对不要仅因为性能而将类、方法等声明成final，声明成final的类、方法一定要确信不再被继承或重载！

* 不需要重新赋值的变量（包括类变量、实例变量、局部变量）声明成final
* 所有方法参数声明成final
* 私有(private)方法不需要声明成final
* 若方法确定不会被继承，则声明成final

## 11.7 垃圾收集和资源释放

* 不要过分依赖JVM的垃圾收集机制，因为你无法预测和知道JVM在什么时候运行GC。
* 尽可能早的释放资源，不再使用的资源请立即释放。
* 可能有异常的操作时必须在try的finally块中释放资源，如数据库连接、IO操作等：

|  |
| --- |
| Connection conn = null;  try{  //do something  }catch(Exception e){ //异常捕捉和处理  e.printStackTrack();  }finally{  //判断conn等是否为null  if(null != conn){  conn.close();  }  }//end try...catch...finally |

# 第12章 可移植性

* 尽量不要使用已经被标为不赞成使用的(deprecated)类或方法。
* 代码应遵循UNIX风格。
* 如果需要换行的话，尽量用 println 来代替在字符串中使用"\n"。
* 用separator()方法代替路径中的“/”或“\” 。
* 用pathSeptarator()方法代替路径中的“:”或“;”。

# 第13章 日志组件

Log中的方法使用：

* log.debug()是放在代码行中间，用来调试时输出使用，永久保留，不用删除
* log.error()是放在try{}catch(){}中catch部分里的用来输出error 异常信息使用的。

注意：日志输出尽量多用log4j，少使用System.out.println();

# 第14章 附录

## CVS标识符

|  |  |
| --- | --- |
| 标识符 | 说明 |
| $Id$ | 标识，包括文件名，版本，提交时间，提交者等信息  $Id: HibernateFilter.java,v 1.1 2003/11/17 13:20:07 l\_walker Exp $ |
| $Author$ | 最后提交者  $Author: l\_walker $ |
| $Reversion$ | 当前版本（服务器版本）  $Revision: 1.1 $ |
| $Date$ | 最后一次提交时间  $Date: 2003/11/17 13:20:07 $ |

## 14.2 常用缩写简表

|  |  |  |
| --- | --- | --- |
| 缩写 | 全字 | 说明 |
| app | application | 应用程序 |
| ctx | context | 上下文信息 |
| mgr | manager | 管理者，经理 |
| conn | connection | 连接（数据库连接） |
| sys | system | 系统 |
| impl | implement | 实现 |